对一类动态规划问题的研究

湖南省长沙市第一中学　徐源盛

【关键字】

动态规划　　费用提前计算　　假设未来决策

【摘要】

本文通过四道题目探讨了一种比较特殊的动态规划问题，即当前决策影响未来“行动”的费用。如果当前决策对未来的影响只与当前决策有关，则直接将对未来费用的影响，算作当前的决策费用计算，并通过状态传递；如果对未来的影响还与未来的情况有关，则新增状态假设未来的情况，待到未来决策时直接使用假设的状态。这就是本论文详细阐述的解题方法。

【正文】

在常规动态规划问题中，我们面临当前状态时“行动”造成的花费往往与这个状态是同时计算的。例如在经典的石子合并问题中，规划方程为 *f[i][j]=max{f[i][k]+f[k+1][j]}+w(i,j)*，当我们计算 *f[i][j]*时，才会把将 *i* 到 *j* 的石子全部合并到一起这一“行动”的费用加进去。这很符合我们的思维习惯。

然而近年来频繁出现一类动态规划问题，在这类问题中，当前“行动”的费用的一部分需要在之前决策时被计算并以状态的形式对当前状态造成影响。造成这一独特的计算的原因就是当前的决策会对未来的“行动”费用造成影响。这类问题构造方程往往比较困难，需要仔细分析原题，找到矛盾所在。

一、 当前决策对未来“行动”的费用影响只与当前决策有关

比如在两男两女中选一男一女去执行任务，已知每个人的效率，希望总效率最高。并且男 A 如果被选，所有女生的效率加 7，如果男 B 被选，所有女生的效率减 7。在第一阶段选男 A 还是男 B 对第二阶段女生的效率有不同的影响，可以将对女生的影响当做男生的“自身魅力”，即把男 A 的效率加 7，男 B 的效率减 7。而我们实际上是把第二阶段的费用在第一阶段计算了。对于这类问题我们往往将对未来“行动”的影响一并算做当前决策的费用。下面通过两道例题进行详细阐述。

【问题一】Sue 的小球(sdtsc 2008)

Sue 和 Sandy 最近迷上了一个电脑游戏，这个游戏的故事发在美丽神秘并且充满刺激的大海上。Sue 有一支轻便小巧的小船，然而，Sue 的目标并不是当一

个海盗，而是要收集空中漂浮的彩蛋。Sue 有一个秘密武器，只要她将小船划到一个彩蛋的正下方，然后使用秘密武器便可以在瞬间收集到这个彩蛋。然而，彩蛋有一个魅力值，这个魅力值会随着彩蛋在空中降落的时间而降低，Sue 要想得到更多的分数，必须尽量在魅力值高的时候收集这个彩蛋，而如果一个彩蛋掉入海中，它的魅力值将会变成一个负数，但这并不影响 Sue 的兴趣，因为每一个彩蛋都是不同的，Sue 希望收集到所有的彩蛋。

Sandy 就没有 Sue 那么浪漫了，Sandy 希望得到尽可能多的分数，为了解决这个问题，他先将这个游戏抽象成了如下模型：

以 Sue 的初始位置所在水平面作为 x 轴。

一开始空中有 N 个彩蛋，对于第 i 个彩蛋，他的初始位置用整数坐标（x_i, y_i）表示，游戏开始后，它匀速沿 y 轴负方向下落,速度为 v_i 单位距离/单位时间。Sue 的初始位置为(x_0, 0)，Sue 可以沿 x 轴的正方向或负方向移动，Sue 的移动速度是 1 单位距离/单位时间，使用秘密武器得到一个彩蛋是瞬间的，得分为当前彩蛋的 y 坐标的千分之一。

现在，Sue 和 Sandy 请你来帮忙，为了满足 Sue 和 Sandy 各自的目标，你决定在收集到所有彩蛋的基础上，得到的分数最高。

输入文件

第一行为两个整数 N, x_0 用一个空格分隔，表示彩蛋个数与 Sue 的初始位置。

第二行为 N 个整数 x_i，每两个数用一个空格分隔，第 i 个数表示第 i 个彩蛋的初始横坐标。

第三行为 N 个整数 y_i，每两个数用一个空格分隔，第 i 个数表示第 i 个彩蛋的初始纵坐标。

第四行为 N 个整数 v_i，每两个数用一个空格分隔，第 i 个数表示第 i 个彩蛋匀速沿 y 轴负方向下落的的速度。

输出文件

一个实数，保留三位小数，为收集所有彩蛋的基础上，可以得到最高的分数。

样例

输入样例(ball.in 的内容)：

```
3 0
-4 -2 2
22 30 26
1 9 8
```

输出样例(ball.out 的内容)：

```
0.000
```

数据范围

N<=20，对于 30%的数据。

N<=1000，对于 100%的数据。

$-10^4 <= x_i,y_i,v_i <= 10^4$，对于 100%的数据。

【分析】

先把所有的点包括起点按 x 值排序，这样题目就变成从起点出发，每次可以向左或向右走到最近的某个彩蛋，将其射落，设每个彩蛋第一次走到的时刻为 t_i，答案就是$\sum(y_i-t_i*v_i)$，使答案最大。

我们注意到已射落的彩蛋集合是不断增大的，这样很容易想到用 $f1[i][j]$、$f2[i][j]$分别表示从起点出发已射落 i 到 j 这一段彩蛋，当前停留在 i 点、j 点的最大得分。考虑 $f1[i][j]$,即点 i 是当前射击的彩蛋，射击的得分与当前时刻挂钩，但是当前的时刻是不能从 $f1[i][j]$的状态中表示出来的。如果我们再添一维状态表示时间，这样时间复杂度是不允许的。

我们发现上述方法的矛盾其实就在于曾经的行走方案会对当前射击的得分产生影响，我们可以进一步考虑 $f1[i][j]$的求解，由于射击 i 的得分是 y_i-t*v_i，而 t 等于之前每一步决策移动的时间总和，这样我们就可以像前述的男生女生问题一样，把$-t*v_i$在之前的移动中就计算，也就是说每次移动都要把未来会减少的得分计算在内。比如说从 $f1[i][j]$推到 $f1[i-1][j]$，即从 i 走到 $i-1$ 时除了 i 到 j 这一段彩蛋外，其它的彩蛋都在下落，将这丢失的分数一并计算到从 i 走到 $i-1$ 中。由于$-t*v_i$已经在之前决策时计算，所以射击时加上 y_i 即可。

为了方便描述，用 $w[i][j]$表示$\sum(v)-\sum_{k=i}^{j}v[k]$，即除了 i 到 j 这一段的彩蛋外的所有彩蛋下落速度和，那么很容易得到方程：

$$f1[i][j]= y[i]+max\{f1[i+1][j]-(x_{i+1}-x_i)*w[i+1][j],$$

$$f2[i+1][j]-(x_j-x_i)*w[i+1][j]\}$$

$$f2[i][j]= y[j]+max\{f2[i][j-1]-(x_j-x_{j-1})*w[i][j-1],$$

$$f1[i][j-1]-(x_j-x_i)*w[i][j-1]\}$$

答案就是 $max(f1[1][n]，f2[1][n])/1000$。算法的时间复杂度为 $O(n^2)$。至此问题已得到完全解决。

回顾上述解题过程，我们发现射击彩蛋的得分被分成两部分计算了，而全体彩蛋下降只与当前移动的时间有关，这使得我们可以把未来的得分一并计算到当前状态中。

【问题二】名次排序问题 (BOI2007 day1)

已知参赛选手的得分，你的任务是按照得分从高到底给出选手的排名。遗憾的是，保存选手信息的数据结构只支持一种操作，即将一个选手从位置i移动到位置j，该移动不改变其他选手的相对位置，即如果i > j，位置j和位置i-1之间的选手的位置都比原来加1，相反如果 i < j，则位置i+1和位置j之间的选手的位置都比原来减一。上述移动的操作的代价定义为i+j，这里，位置编号从1开始。

请你编程确定一个移动选手的步骤，将选手按照得分从高到低排序，并使整

个移动过程的总代价最小。

输入：

文件sorting.in第一行为一个整数n（2<=n<=1000），表示选手的人数；接下来的n行，每行一个非负整数S_i(0<=S_i<=1000000)，表示一个选手的得分。你可以认为每人的得分是不同的。

输出：

文件sorting.out的第一行为一个整数，表示移动的次数，接下来的每一行表示一个移动步骤，每个移动步骤用两个整数i, j表示，表示位置i的选手移动到位置j，i和j之间有一个空格隔开。

样例

soring.in	sorting.out
5 20 30 5 15 10	2 2 1 3 5

【分析】

为了方便讨论，将每个点的值修改为其目标位置，这样目标状态就是 *1* 到 *n*。假设 *n+1* 号人在 *n+1* 位置。初看此题感到无从下手，稍加分析，不难发现：将 *i* 移动到 *j* 位置有两种方法，即主动移动与被动移动，而且我们感觉上觉得一个人如果移动多次不如一步到位。

[猜想 2.1]一个人最多移动一次，如果移动则必然移动到编号比他大 1 的人前面。

因为往后移动会使部分人的位置减 1，从而减少此人未来的移动费用，所以编号越大的越先移动。

[猜想 2.2]按编号从大到小进行移动。

由于上述猜想的证明不是本论文的重点，在此省略，详细证明在附录中。

[结论 2.1]：按编号从大到小进行操作，对于每个人 *x* 有两种选择:

1. **移动到 *x+1* 前一个位置。**
2. **如果 *x* 在 *x+1* 前，则不移动，以后再将所有处于 *x* 和 *x+1* 之间的移动到 *x* 之前。**

考虑将处于位置 *i* 的 *x* 向后移动到处于 *j(j>i)*的 *x+1* 前。由**[结论 2.1]**我们可知，当前小于 *x+1* 的都没有移动，也就是说**所有小于等于 *x* 的数的相对位置与最初的一样[推论 2.1]**。又因为比 *x+1* 大的数要么在之前移动了，要么 *x+1* 移动到它前面了，所以**所有比 *x+1* 大的数都位于 *x+1* 后面[推论 2.2]**。注意到这一题移动会导致位置的改变，*i* 和 *j* 已经不再是最初 *x* 和 *x+1* 所在的位置了。那我们该怎么办呢？前面提到了相对位置是不会变的，可否在此做文章？

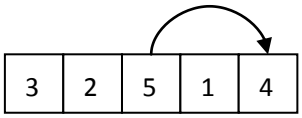

图 2.1

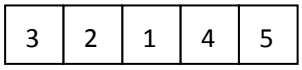

图 2.2

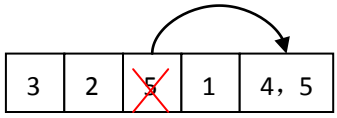

图 2.3

如图 2.1，把 5 放到 4 后面，得到如图 2.2 所示。1 和 4 都向前移动一位，如果我们修改规则，让 5 和 4 共用位置 5 得到图 2.3，从而并没有移动 1 和 4，而此时 1、2、3、4 的相对位置仍然没改变。其实我们可以利用相对位置求出绝对位置！如果 *x* 的原位置为 *p1*，由**[推论 2.1]和[推论 2.2]**可知 *x* 的实际位置为最开始 1 到 *p1* 中比 *x* 小的数的个数加 1（加上自己）。但是 *x+1* 可能移动了，那么在原序列中的位置只能枚举，设为 *p2*。用 *c(p,x)*表示处理完 *x+1* 到 *n* 后 *x* 在原序列中位置 *p*，那么 *x* 的实际位置为 *c(p,x)*。*c(p,x)*等于原序列中从 1 到 *p* 有比 *x* 小的数的个数加 1，那么把 *x* 从原序列的 *p1* 移动到 *x+1* 所在的 *p2* 的费用为 *c(p1,x)+c(p2,x+1)-1*（因为是把 *x* 插入到 *x+1* 前，所以要减 1）。

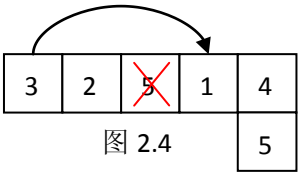

图 2.4

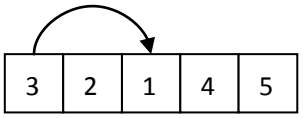

图 2.5

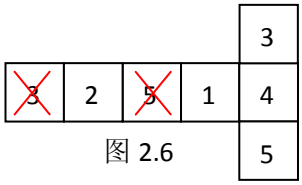

图 2.6

例如，在图 2.4 中，把 3 插入到 4 前面一位，由图 2.5 易知 *i=1,j=4*。3 的实际位置为图 2.4 中在 3 前面且比 3 小的数的个数加 1 为 1，同理 4 的实际位置为 4（5 必然在 4 后面），这次移动的费用为 1+4-1，的确与图 2.5 的实际情况相吻合。为了不影响 1、2 的位置，3 在移动后在原序列中位置变成了 5，真实位置为 3，如图 2.6 所示，也就是说经过这次移动后 3，4，5 的原序列中位置均为 5。

我们可以令 *f[x][p2]*表示把 *x* 移动到原序列的 *p2* 位置且 *x+1* 到 *n* 都在 *p2* 的最小费用。*pos[x]*表示 *x* 在原序列中位置。则对于向后移动的情况，有方程 *f[x][p2]=f[x+1][p2]+c(pos[x],x)+c(p2,x+1)-1(p2>pos[x])*。

接着我们考虑 *j<i* 的情况。如图 2.7 所示，如果当初 4 和 5 都没移动，把 3 移动到 4 之前时，3 的位置并不等于 *c(5,3)*，而是 *c(5,3)+2*；如果当初 5 的决策是移动到最后，而 4 的决策是不移动，如图 2.8，也就是 4 在原序列中位置仍为 2，这样把 3 移动到 4 之前时，3 的位置是 *c(5,3)+1*。也就是说过去的决策对当前 3 的移动费用造成了影响。不妨仿造问题一的处理方法，将对未来的影响一并计算到当前状态，这样 *j<i* 的情况可以归于 *j>i* 的情况。

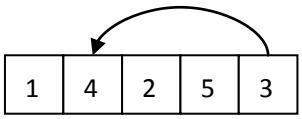

图 2.7

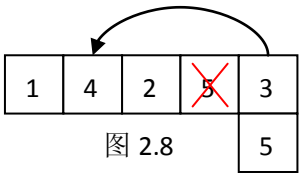

图 2.8

下面我们考虑如何计算对未来状态的影响。

如果 *x* 不移动，则 *x+1* 必然在 *x* 后面的某个位置，设在原序列中位置分别为 *p1,p2*，*p1=pos[x]*。还是考虑图 2.7，当我们决定 5 不移动时，如果 4 移动到 5 的前面，或者 4 不移动， 3 都要跨越 4 和 5。也就是说 3 在未来移动时位置为

c(5,3)+(5-3)，而 *c(5,3)*这部分费用会在将来的移动中被计算，当前需要增加的仅有*(5-3)*。总而言之，如果我决定 ***x*** 不移动，虽然这一步并不产生费用，但是要将未来的一部分必然会产生的费用计算，那就是**所有位置大于 *pos[x]*小于 *p2* 的比 *x* 小的数与 *x* 差的和**。用方程表示：

$$f[x][pos[x]]=min\{f[x+1][p2]+\sum_{k=pos[x]+1}^{p2-1}(x-s[k])x>s[k]\}p2>pos[x]$$

综上所述，此题的方程为：

$$f[i][j]=f[i+1][j]+c(pos[i],i)+1+c(j,i)+1$$

$$f[i][pos[i]]=min\{f[i][pos[i]],f[i+1][j]+\sum_{k=pos[i]+1}^{j-1}(i-s[k])i>s[k]\}j>pos[i]\}$$

答案就是 $min\{f[1][i]\}1<=i<=n$。时间复杂度 $O(n^2)$。

【小结】

回顾第一题的解题历程，我们先发现当前“行动”的费用受到过去决策的影响，如果新增状态 *t* 表示过去决策的影响，状态数过大将会无法承受，于是我们将受到的影响在过去决策时计算，通过状态传递过来。

我们之所以能这样做是有条件的：

一、影响是必然的。在问题一中，只要我们一移动，彩蛋就会下降。在问题二中，只要我决定不移动，未来的费用就会增加一个固定值。如果问题一不要求射击所有彩球，那么未来不会被射击到的彩球就不会被影响，因而就不能使用上述方法解决。

二、影响是关于当前决策的函数。问题一中的函数是*-t*∑v*，*∑v* 与状态有关，可以看成常数，*-t* 即当前决策的移动时间。问题二的函数同样满足。

以上两个条件综合来讲就是**当前决策对未来“行动”费用的影响只与当前决策有关。**

二、 当前决策对未来“行动”的费用影响不只与当前决策有关

有些问题并非像上述问题这么简单，当前决策对未来“行动”费用的影响还有可能与未来的情况有关。这时我们往往多开几维状态来假设未来的决策，从而像之前那样将部分费用提前计算，待到未来决策时即可直接使用。

【问题三】方块消除(算法艺术与信息学竞赛)

Jimmy 最近买了台电脑，很快，像大多数孩子一样，他迷上了游戏。最近，他正在玩一款叫做方块消除的游戏。游戏规则如下：

如图，n 个带颜色的方格排成一行。

相同颜色的方块就会连成一个区域（如果两个相邻方块颜色相同则这两个方块属于同一区域）。你可以任选一个区域消去。设这个区域包含的方块数为 *x*，则你将得到 x^2 的分值。方块消去之后，其右边的方块会往左移动，与左边的相

连。

【分析】

我们可以先把同色的合并，假设用*(a,b)*表示有 *b* 个 *a* 颜色的方块相连。

每次消除一个区域，加上这个区域的得分，这很像石子合并问题，很容易让人尝试用 *f[i][j]*表示把区域 *i* 到区域 *j* 的方块消除完的最大得分。考虑区域 *j*，如果单独消除，那么 $f[i][j]=f[i][j-1]+b_j^2$。还有一种情况，区域 *j* 与之前的某些同色的方块合并成新区域再消除。可是必须知道合并后的长度才能计算得分，但是仅从当前的状态是无法得知的，如果我们新增状态假设过去的情况，那么我们要表示过去的每一块方块是否消除，新增状态数多到无法承受。

假设我们还像前面两题那样的方法去解决，假设当前消去一个长 3 的红色方块，过去留了一个长 3 的红色方块，总共消去得分是 $6^2=36$，但是 36 并不能像上面两题那样进行拆分计算，究其原因，这是一个二次函数，在过去留下长 3 的红色方块时对我们现在一起消去的影响与我们现在消去的长度是有关的。

这时我们只能换个思路，先理清我们要做的：我们要把区域 *j* 与之前的某些同色方块合并，如果这次合并在之前被料到了并预先计算了得分，并通过状态传递到了现在，那我们不就能构造出一个动态规划方程了么？也就是说对于当前的 *f[i][j]*我们还要假设经过一些消除操作后，有 *k* 个与 a_j 同色的方块连在了区域 *j* 后形成新的长度为（b_j+k）的区域，即用 *f[i][j][k]*表示将 *i* 到 *j* 合并，并且假设未来会有 *k* 个与 a_j 同色的方块与 *j* 相连的最大得分。为什么只要记录 *j* 在未来会有多少个与之相连，而不用记录 *i* 到 *j-1* 的呢？

[证明 3.1]

如图 3.1 所示，如果区域 *j* 未来会与 k_2 相连，也就是说 *j+1* 到 k_2*-1* 之间的区域需要先于区域 *j* 被消掉，所以区域 *j* 之前的区域必然不能与 *j* 到 k_2 之间的区域相连。如图 3.2 所示，如果 *j* 未来会与 k_1 相连，*j* 之前的某点 *i* 会与 k_2 相连，那么我们可以把消去 *i* 的得分在计算 *f[i][*k_2*]*时加进去。

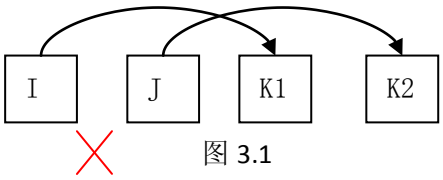

图 3.1

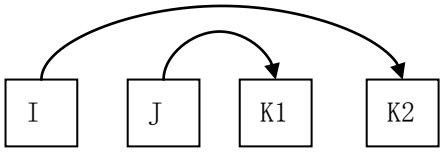

图 3.2

于是我们得到**[结论 3.1]**：对于状态 *f[i][j][k]*，不论经过什么操作,区域 *i* 到区域 *j-1* 之间的区域只能与区域 *i* 到区域 *j* 之间的区域相连，而区域 *j* 可以与 *j* 之后的区域相连。

因此我们得到方程：

（1） 如果直接消去第 *j* 个区域和未来会接到 *j* 后面的 *k* 块，那么 $f[i][j][k]=f[i][j-1][0]+(b_j+k)^2$。

（2） 如果 *j* 与之前一起合并，假设与 *j* 颜色相同的是区域 *p*，那么 $f[i][j][k]=f[i][p][k+b_j]+f[p+1][j-1][0]$（$i<=p<j$ 且 $a_p=a_j$）

*f[i][j][k]*为两者的最大值。

答案即是 *f[1][n][0]*。时间复杂度为 $O(n^4)$。本来我们想将计费摊在之前每一

次“保留，以后消除”的决策上，但是我们发现这些计费不是独立的，是相互关联的，当前的费用还需知道未来的情况，所以我们可以再开一维状态，用以记录未来的情况。

【问题四】奥运物流(NOI 2008 day2)

2008 北京奥运会即将开幕，举国上下都在为这一盛事做好准备。为了高效率、成功地举办奥运会，对物流系统进行规划是必不可少的。物流系统由若干物流基站组成，以1…*N* 进行编号。每个物流基站*i*都有且仅有一个后继基站*Si*，而可以有多个前驱基站。基站*i* 中需要继续运输的物资都将被运往后继基站*Si*，显然一个物流基站的后继基站不能是其本身。编号为1 的物流基站称为控制基站，从任何物流基站都可将物资运往控制基站。注意控制基站也有后继基站，以便在需要时进行物资的流通。在物流系统中，高可靠性与低成本是主要设计目。对于基站*i*，我们定义其“可靠性”*R*(*i*)如下：设物流基站*i* 有*w* 个前驱基站$P_1,P_2,P_3…P_w$即这些基站以*i* 为后继基站，则基
站 *i* 的可靠性 *R*(*i*)满足下式：

$$R(i)=C_i+k\sum_{j=1}^{w}R(P_j)$$

其中C_i和*k*都是常实数且恒为正，且有*k*小于1。整个系统的可靠性与控制基站的可靠性正相关，我们的目标是通过修改物流系统，即更改某些基站的后继基站，使得控制基站的可靠性*R*(1)尽量大。但由于经费限制，最多只能修改*m* 个基站的后继基站，并且，控制基站的后继基站不可被修改。因而我们所面临的问题就是，如何修改不超过*m* 个基站的后继，使得控制基站的可靠性*R*(1)最大化。

输入格式

输入文件 trans.in 第一行包含两个整数与一个实数，*N*, *m*, *k*。其中*N* 表示基站数目，*m* 表示最多可修改的后继基站数目，*k* 分别为可靠性定义中的常数。

第二行包含*N* 个整数，分别是$S_1, S_2…S_N$，即每一个基站的后继基站编号。

第三行包含*N* 个正实数，分别是$C_1, C_2…C_N$，为可靠性定义中的常数。

输出格式

输出文件 trans.out 仅包含一个实数，为可得到的最大*R*(1)。精确到小数点两位。

输入样例

4 1 0.5
2 3 1 3
10.0 10.0 10.0 10.0

输出样例

30.00

【分析】

将每个点向它的后继发出一条边，稍加分析不难得出，此题的图是一些树，

树的根节点连成一个环。

对于一棵以*i*为根的树，通过方程的迭代，我们发现*R(i)*等于所有以*i*为根的树中包含的点*x*的可靠常数$C_x*k^{d(x,i)}$的和。*d(x,i)*表示*x*到*i*经过的边数。

考虑环，如图4.1，一个长度为3的环。$R(1)=C_1+kR(2)$。$R(2)=C_2+kR(3)$。$R(3)=C_3+kR(1)$。消元得$R(1)=C_1+kC_2+k^2C_3+k^3R(1)$。$R(1)=(C_1+kC_2+k^2C_3)/(1-k^3)$。即环上每个点的*x*的可靠常数$C_x*k^{d(x,1)}$的和除以$(1-k^{len})$，*Len*表示环的长度。

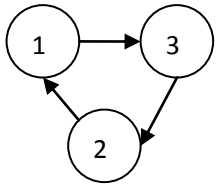

图 4.1

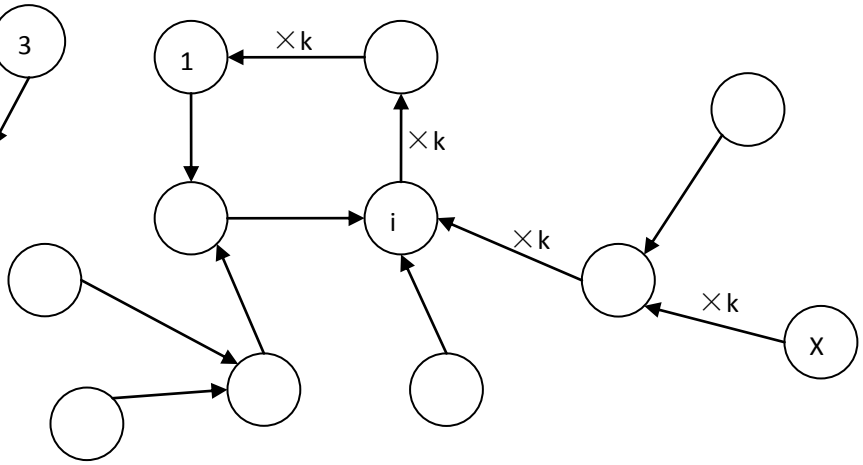

图 4.2

如图4.2所示，每个点*x*对*R(1)*的贡献为$C_x*k^{d(x,i)}*k^{d(i,1)}=C_x*k^{d(x,1)}$。于是我们得到：

$R(1)=\sum_{j=1}^{n} ci\times k^{d(i,1)}/(1-k^{len})$。我们枚举环的长度，即修改环上某个点的后继，这样$1-k^{len}$确定，只需分子尽可能大。现在我们有*m*次修改机会，因为环已经确定，所以不能再修改树根的后继，只能改变一个非树根点的后继。根据公式，显然每次修改都应该把修改点的后继直接设置为1。

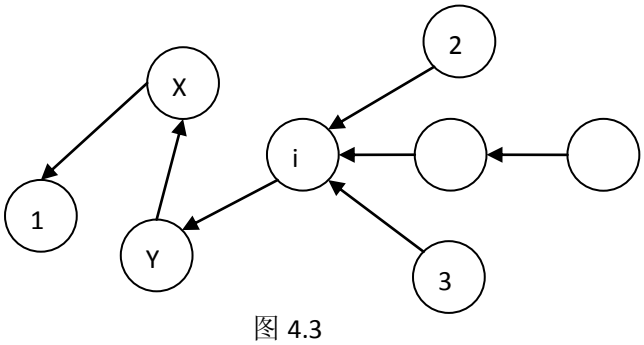

图 4.3

尝试使用动态规划，如图4.3，我们很容易想到，用在以*i*为根的树中，修改了*j*次的最大值来表示状态。对于点*i*我们有修改和不修改两种方式。

如果不修改*i*的后继，我们要把*j*次修改分配到*i*的子树中，然后加上*i*在当前状态下对1的贡献$C_i*k^{d(i,1)}$。

如果将*i*的后继修改为1，*i*为根的树对*R(1)*的贡献怎么计算呢？我们发现，假如*i*的子孙都没修改过，将点*i*的后继设置成点1，那么点i及点i的子孙到点1的距离都减少了2，即贡献值都除以k^2。但是如果点2的后继在之前已经被设置成了1，

那当我们把点*i*的后继修改成点1的时候点2的距离并没有减少2。**也就是说点2的决策影响着改变点*i*后继的费用。**难道我们新增状态表示树*i*中哪些点被修改过，哪些点没被修改过？显然状态量是无法承受的。

我们需要把点2对修改点i时的费用贡献在决策点2的时候计算。即当我们决策点2时，加上未来的费用贡献。可是我们应该增加多少呢？如果点*i*后继修改为点1，点2的距离为2。如果点i没有修改而是点*y*修改到点1，点2的距离是3，点2的贡献取决于离它最近的被修改的祖先，需要新增状态表示离它最近的被修改的祖先。用*f[i][j][d]*表示以点*i*为根的树中，修改*j*次，且点i到1的距离为*d*的最大值。

考虑i的儿子2，*d(2)*要么为1(修改2的后继)，要么为*d(i)+1*(不修改2的后继)。可令*g[i][j][d]=max{f[i][j][d+1],f[i][j][1]}*

那么我们可以得到方程:

(1).如果*i*不修改后继,那么除非点i本身到1的距离为1，否则*d*不能为1。

$f[i][j][d]=max\{g[s_1][j_1][d]+g[s_2][j_2][d]+\ldots+g[s_t][j_t][d]\}+c[i]*k^d$

$(j_1+j_2+\ldots j_t=j)s_1,s_2\ldots s_t$为*i*的*t*个儿子。对于*max*可以再进行一次动态规划，用*FF[i][j]*表示前*i*个儿子分配了*j*次修改的最大贡献，$FF[i][j]=FF[i-1][k]+g[s_i][j-k][d]$，*max*函数的最大值为*FF[t][j]*。

(2).如果*i*修改后继

$f[i][j][1]=max\{g[s_1][j_1][1]+g[s_2][j_2][1]+\ldots+g[s_t][j_t][1]\}+c[i]*k$

$(j_1+j_2+\ldots j_t=j-1)s_1,s_2\ldots s_t$为*i*的*t*个儿子。

最后考虑环的情况，由于环已经枚举确定了，所以答案就是：

$max\{f[p_1][j_1][0]+f[p_2][j_2][1]+\ldots+f[p_{len}][j_{len}][len-1]\}(j_1+j_2+\ldots j_{len}=m)$ p_i为环上的点，如图4.1答案就是$max\{f[1][j_1][0]+f[2][j_2][1]+f[3][j_3][2]\}(j_1+j_2+j_3=m)$。这里可以再用一次动态规划解决。

因为每个点只会被背包计算一次，所以时间复杂度是$O(len*n^2*m^2)$。因此可以在时限内出解。近年与此题类似的题目大量涌现，如IOI2005《河流》、NOI2006《网络收费》，这类题都是在树上进行动态规划，并且将本应该在点i计算的费用转化到点i的子孙上。在规划i的子孙时假设未来的情况，并以状态的形式记录下来，例如这道题是假设最近的修改点的位置，等到规划点i时直接使用过去假设的决策。

【小结】

这类问题可以说是第一类问题的复杂版，同样是发现过去决策影响当前“行动”的费用，同样是发现如果在当前状态记录过去的决策会使状态数过多，只能将这部分影响在过去计算。不过，当前决策对未来“行动”费用的影响又与未来情况有关，这时我们假设未来的情况，将不同的影响沿着不同的状态传递到未来。

三、 总结

本文的四个例子，方程构造都是有一定难度的，其原因就在于当前“行动”的费用受到过去决策的影响，而如果新增状态在当前假设过去的情况，状态数过多。于是我们改变“时间观”，从过去考虑当前，即从当前考虑未来，把当前决

策对未来的影响计算到当前状态。

对于第一类问题，只需把对未来的影响算作当前决策的费用，保存在当前状态。而对于第二类问题，需要假设未来的情况，把不同未来情况的影响保存在不同的状态中，可以理解为把影响沿着不同的路传递到未来，待到未来决策时直接使用。

当然，这两类问题变化是无穷无尽的，但只要我们善于发现，勇于创新，就能构造出方程。

【感谢】

衷心感谢曹利国、周祖松老师在写这篇论文时给我的指导和帮助。

衷心感谢寻云波、李远韬、吴空、易茜、邓方丁、吴一凡等等同学给我的帮助。

【参考文献】

[A]《算法艺术与信息学竞赛》——刘汝佳、黄亮，清华大学出版社
[B]WC2008孙辉讲稿
[C]BOI2007官方题解

【附录】

【附录 A】

关于**[猜想 2.1]**、**[猜想 2.2]**的证明，来自 BOI2007 官方题解：

Lets look at the player pmin with the smallest score, who has to end up at position n.

Claim 1: If pmin is moved at all, it is optimal to move him directly to the end of the list.

Proof: Obviously, moving him towards the front will only increase the costs of other players to be moved. So assume he will be moved towards the end to a position i < n (thus saving n − i steps). Now there are two possibilities: either, the n − i players after him in the list have to be moved and their costs are increased by 1 (as opposed to that pmin was moved to the end), or pmin has to be moved again, which clearly is not optimal.

Claim 2: If pmin is moved at all, we can do this in the first move.

Proof: Following from Claim 1, we already know the second part of the moving costs is fixed. So the only reason not to move pmin first is that his current position decreases caused by moving other players. But these players have their moving costs increased by 1 because pmin wasn’t moved to the end.

If pmin is moved, we have reduced our problem to the same problem

with n − 1 players. If pmin is not moved, we have a similar problem with n – 1 players with the additional restriction that all players after pmin have to be moved before pmin.

Let p0min be the player with the smallest score among the remaining n – 1 players. We can see that it does not help to move p0min after pmin, because that would mean we have to move him again, and we gain nothing from moving him past other players still to be moved (for each of them, we decreased the moving cost by 1, and increased the moving cost of p0min by at least one). If p0min is currently placed before pmin in the list, by similar arguments as for Claim 1 we can argue that if p0min is moved at all, it is optimal to move him directly before pmin. Also, Claim 2 still works. So we are left with the case that p0min is currently placed after pmin and has to be moved towards the front. If there are players between pmin and p0min, we could consider moving them first. If we move p0min first, the start positions of the players in between are increased by 1. But if we move some player in between first, the end position where p0min has to be moved to increases by 1, so it can't be bad if we move p0min first.

So, we can see that by using induction and the arguments presented above we can prove that each player is moved at most once, and the players can be moved in ascending order according to their score.

【附录 B】

问题一源程序：

Ball.cpp

问题二源程序：

Sorting.cpp

问题四源程序：

Trans.cpp